SOFTWARE MAINTENANCE: RESEARCH AND PRACTICE, VOL. 9, 329–342 (1997)

Practice

Using COBOL Defensive Traps

ERIC GARRIGUE VESELY*
Analyst Workbench Consulting Sdn. Bhd., 46000 Petaling Jaya, Selangor Darul Ehsan, Malaysia

SUMMARY

This paper focuses on inserting defensive traps in COBOL programs during maintenance in order to improve the quality of the software. The paper defines 'defensive trap' and describes two types, active and passive. One of the major opportunities present during maintenance is to fix omissions of defensive traps by inserting them into the software. The paper describes nine situations, involving arithmetic verbs, CALL, STRING, UNSTRING, EVALUATE, IF **and** ELSE, GO TO DEPENDING, SEARCH**, and input and output verbs. The other major opportunity present during maintenance is to correct for the weak use or misuse of defensive traps. The paper describes five situations, involving index and subscript checks,** IF NEGATIVE**, overlapping operands,** PERFORM **and** EXIT**, and qualification. The paper includes some examples from experience with inserting and strengthening defensive traps. © 1997 John Wiley & Sons, Ltd.**

J. Softw. Maint., **9**, 329–342 (1997)

No. of Figures: 4. No. of Tables: 0. No. of References: 8.

KEY WORDS: software quality improvement; COBOL usage; isolation techniques; domain testing; arithmetic operations; program control flow

1. INTRODUCTION

Maintenance programmers and analysts can improve the quality of the COBOL software they maintain by adding and strengthening defensive traps in the COBOL source code. Statistics and personal reviews of thousands of COBOL programs show that virtually all programs have at least one missing or misused defensive trap. Each missing or misused trap acts as a defect, providing an opportunity for data corruption to happen and for runs to produce incorrect results—and work for maintenance personnel (Vesely, 1989b).

Unlike hardware traps (Pollack and Sterling, 1993) 'defensive traps' are sets of statements incorporated into the program source code for the purpose of detecting and handling out-of-domain data and for detecting operations that are illegal for the type of data to be operated upon. Two examples of out-of-domain data are the date 20011244 (44 January 2001), and for a cheque to be paid, a negative number (such as –\$1.00) as the face amount. Two examples of operations illegal for a data type are adding the

* Correspondence to: Eric Garrigue Vesely, Analyst Workbench Consulting Sdn., 272A, Jalan 5/51, Petaling Garden, 46000 Petaling Jaya, Selangor Darul Ehsan, Malaysia. E-mail: amythai@jaring.my

CCC 1040–550X/97/050329–14\$17.50

Received 13 November 1993
Revised 24 January 1995

alphabetic literals ‘CAT’ and ‘DOG’, and comparing two numbers, one of them signed and the other unsigned.

Typically, the statements comprising a defensive trap act to detect data values that are outside of the set of legal allowable values, and then act to prevent or limit the consequences of the presence of the invalid data. The defensive posture taken is that all data are potentially out of domain until proven to be within domain, and only data within domain are to be processed. In batch-orientated delayed-time systems, the need for defensive traps was less than it is today because often the operations or user personnel could identify the data with the incorrect values, correct the values, and resubmit the run. In modern OLTP (on-line transaction processing) systems and in nearly all real-time systems, the costs of incompetent or missing defensive traps can range from user or customer annoyance to sudden catastrophic losses to property and human life.

Defensive traps can be proactive in that some act (are executed) before an exception or unwanted condition affects the process or function. Defensive traps supplement and add to, but are not a substitute for, the error checking provided automatically by a runtime environment, an operating system or a compiler.

COBOL maintenance programmers and analysts have many verbs, clauses and phrases to trap unusual, exception or error conditions. Figure 1 lists most of the common ones.

Also, programmers and analysts may use programmatic traps as defensive traps. A crude example is placing DIVIDE BY ZERO as the last statement of a GO TO DEPENDING ON to trap error conditions, such as when the DEPENDING ON value exceeds its domain. DIVIDE BY ZERO in most runtime environments abends (aborts the execution of) a program, sometimes sending control to a destination that produces a formatted dump of internal storage. However, some users (customers) because of their needs and expectations regard permitting a program to abend to be an unacceptable way of handling out-of-domain data.

Fortunately, triggering most traps does not result in abends. The possibility of abends makes inserting traps be controversial in some organizations for some types of applications. Suppose a trap is inserted into a program and then later the program abends because the trap got triggered during a normal production run. This means that the program has probably been producing some corrupt data or incorrect results, or both, for a long time, probably years. Why did the data users not notice that, and does the Information Systems organization now need this kind of publicity, let alone the abend?

Inserting traps also complicates regression testing, since the run results of a program with inserted defensive traps do not always equal those of the same program without the added or changed defensive traps. This then requires work to affirm that the original legacy (untrapped) program was wrong and the now trapped-equipped version of this legacy program is right. Who pays for that work and why?

```
ON SIZE ERROR        ON OVERFLOW          WHEN OTHER
INVALID KEY          ON EXCEPTION         FILE STATUS
IF...relation        DEBUGGING            ELSE
AT END               USE AFTER STANDARD EXCEPTION
```

Figure 1. A list of some common aids for doing defensive trapping in COBOL

2. DEFENSIVE TRAP TYPES

2.1. Passive traps

Passive defensive traps have less impact on performance (execution speed) since they only trigger when an anomaly occurs. An example of a passive trap is ON SIZE ERROR which only triggers when unintended truncation occurs. The following passive traps should be inserted during maintenance, if missing (Vesely, 1991), provided their side-effects are more acceptable than are the consequences of the conditions they trap:

- ON SIZE ERROR into arithmetic statements,
- ON EXCEPTION into CALL statements,
- ON OVERFLOW into STRING and UNSTRING statements,
- WHEN OTHER into EVALUATE statements,
- DIVIDE BY ZERO into IF statements,
- DIVIDE BY ZERO into GO TO DEPENDING statements,
- AT END into SEARCH statements,
- FILE STATUS or equivalent into input/output statements, and
- runaway abend paragraphs after each EXIT paragraph.

2.2. Active traps

Active defensive traps may impact performance because additional object code must be executed frequently. For that reason, active traps are normally switch controlled. Usually, they are left activated (turned on) during testing or debugging runs, and turned off otherwise. An example of an active trap is checking subscripts for domain conformance (for being within table range). This always requires additional object code execution each time the subscript is used.

The normal way to deactivate active traps is to place D in column 7 of their statements, or * in column 7 for WITH DEBUGGING MODE. The * turns DEBUG off and D statements become comments.

The following active traps should be inserted during maintenance, if missing (Vesely, 1989a), provided their side-effects are more acceptable than are the consequences of the conditions they trap:

- index and subscript domain checks,
- IF NEGATIVE test on data from unsigned numeric data involved in MOVE or SUBTRACT statements,
- overlapping operand checks on any data from within 01 level data definitions, and
- establish counters to detect incorrect invocations of active transfers of control, such as when using PERFORM.

3. FIXING OMISSIONS OF DEFENSIVE TRAPS

3.1. Arithmetic verbs

All COBOL arithmetic verbs have an optional ON SIZE ERROR phrase as a means of compensating for weakness in prior domain checking. When an arithmetic operation, for example, addition or division, generates results that do not fit properly within the receiving operand as defined, the operation may not complete. Then, runtime environment detects a size error and permits the execution of statements to continue, but leaves the values in affected operands *undefined*. That is, the data values may have been altered, but not necessarily in a way conforming with what the programmer or analyst would expect from a correct execution of the arithmetic function. In other words, the values in the affected operands may become corrupted. By including an *ON SIZE ERROR*, the analyst or programmer can specify corrective or compensating action.

Usually, the best fix is to correct the data values of one or more of the operands. This is usually a difficult process because the equivalent correction has to be made in all programs that use these same operands. Cross-reference (xref) listings can usually reveal the presence of the operands. However, the offending operand values may be correct representations of the transaction or situation. In such cases, the best fix is to reformulate the ‘business rules’ embodied in the program and incorporate explicit domain checking in advance, and the ON SIZE ERROR as backup protection.

A quick fix is to WRITE the operand data items to an error report, and bypass the further processing of the transaction or situation data. This quick fix is decreasing in popularity as on-line and real-time applications increase in importance, and as the labour cost to introduce correction data and make the reruns continues to increase. The labour cost arises partly because the offending operand data may indicate changes in the work done in the organization, where those changes have not yet been adequately reflected in the existing software systems. Size errors can arise from clerical errors, but may be symptoms of degradation in the fit of the software system to the needs of its users. If so, quick fixes are not appropriate.

An international subsidiary of a telecommunications company kept records of and processed data about telephone calls initiated, calls completed, call durations, etc. The COBOL software for this work had been written in the late 1970s, and the data division had been left largely untouched since then. Recently, the frequency of user complaints about inconsistent call statistics had been increasing. Output reports that indicated both that call volume was increasing and decreasing attracted management attention. Since the software was largely lacking defensive traps, ON SIZE ERROR traps were inserted as a first step. The first run thereafter generated hundreds of trap triggers. The analysts and programmers updated the data definitions in the data division for the arithmetic data items, and since then, the user complaints about inconsistencies have dropped to zero.

3.2. CALL

The usage of CALL has varied greatly from organization to organization. Some very rarely use CALL, whereas others use it extensively. When used, it should be used with

its optional ON OVERFLOW or ON EXCEPTION phrase. Failure to include either means that if the called (invoked) program is unavailable, then the calling (invoking) continues with the next executable statement. When that happens, whatever the called program was supposed to do did not get done. One fix is to have the called program set the value of a success/failure indicator field (return a value) that the calling program has initialized to indicate failure, and then test the value of that field in the statement immediately following the CALL in the calling program.

3.3. STRING

STRING joins the partial or complete contents of one or more sending operand data items into a single receiving operand data item. STRING has an optional POINTER phrase used in moving the partial contents. When (POINTER $<$ 1) or (POINTER $>$ maximum-receiving-size) then overflow occurs during execution. Hence, the failure to include a defensive trap testing the magnitude of the pointer can result in data corruption and the passing of control to the next executable statement. One fix is to include such a defensive trap. Another is to insert an optional ON OVERFLOW phrase into the STRING statement. Another is to replace the STRING with a set of MOVE statements supported by appropriate data descriptions.

3.4. UNSTRING

UNSTRING causes contiguous data in the sending operand to be separated into pieces, and the pieces copied into specified receiving operand data items. Like STRING, UNSTRING has an optional POINTER phrase used in moving the partial contents. When (POINTER $<$ 1) or (POINTER $>$ maximum-receiving-size) for each part, then overflow occurs during execution. Overflow also occurs when data are left still to be copied, yet there are no more receiving data items available. Hence, the failure to include a defensive trap testing the magnitude of the pointer can result in data corruption and the passing of control to the next executable statement. The fixes are parallel to those noted for STRING.

3.5. EVALUATE

EVALUATE is a multi-branch multi-join statement that tests multiple conditions and that has an optional WHEN OTHER phrase. When an execution uses data that fail to satisfy any of the specified conditions, control falls through to the next executable statement. However, if the WHEN OTHER phrase is present, then control goes to it instead of falling through. Hence, the usual fix is to insert a WHEN OTHER phrase as a catcher for the non-specified conditions. A less satisfactory fix is to WRITE the condition data items to an error report, and bypass the further processing of the transaction or situation data. This fix suffers from the difficulties noted earlier for arithmetic verbs.

3.6. IF and ELSE

Every IF test is actually two-sided, with the execution of only one of the sides linked to satisfying specified conditions. While COBOL provides an optional ELSE to go with

the IF, COBOL does not permit the programmer to specify explicitly the conditions to be satisfied for the ELSE action to be executed. COBOL leaves them to be inferred. When the specified conditions are not satisfied during an execution and no ELSE is present, control falls through to the next executable statement, as if no IF had been incorporated into the program. This sometimes gives unwanted results. The usual fix is to pair each IF with an ELSE, and while it does not affect execution, add comments explaining the decision-making that the IF and ELSE implement.

The situation can be complicated with multiple conditions when logical AND, OR or NOT are part of the IF statement. While these have scant effect upon computer execution speed, they slow programmer and analyst understanding speed during maintenance. AND is the equivalent of IF nested within IF, while OR is the equivalent of IF nested within ELSE. This can be used as a fix to restructure complex IF statements at the cost of adding lines of source code to the program. The usual fix for reducing or eliminating NOT is to change the direction of a test, such as ⟨= instead of NOT ⟩. A weak fix for IF statements is to insert one or more PERFORM statements to WRITE error data and bypass some processing. An unpopular fix is to insert a DIVIDE BY ZERO phrase. Both of these two fixes suffer from the difficulties noted under arithmetic statements.

3.7. GO TO DEPENDING ON

The most common use of GO TO DEPENDING ON is for implementing case structures by using PERFORM statements. The execution of each set of performed code occurs when its specified set of conditions is satisfied. If any of the condition data values are out of domain or signed or fractional, then the runtime environment ignores the GO TO DEPENDING ON statement and continues with the next executable statement. The best fix is to pretest the condition operands against their domains. A more expensive and cumbersome fix is to rewrite the GO TO DEPENDING ON as a set of IF statements. A quick fix is to insert a DIVIDE BY ZERO phrase as described earlier.

3.8. SEARCH

SEARCH locates specific data in a table, with the ability to make the process depend on conditions specified with optional WHEN phrase. When in execution none of the specified conditions are satisfied and the optional AT END phrase is omitted, control falls through to the next executable statement. The easy fix is to insert an AT END phrase, but data corruption of the INDEX operand may still occur.

Because of the wealth of options available with SEARCH, additional defensive traps may be helpful. A SEARCH can be terminated by inserting an END SEARCH phrase. Also when the SEARCH is nested within an IF statement, a SEARCH can be terminated by an END-IF used as the terminator for an IF statement. When the ALL option is used, the possibilities of data corruption increase. The usual cause is out of domain values for operands used in one or more of the WHEN phrases, or an out-of-domain sequencing (ordering) of the data in the table. Neither of these have quick fixes; the best fix is domain checking in advance of executing the SEARCH. The WRITE and bypass fix described for arithmetic verbs can be used with SEARCH.

The inland tax department of a governmental unit had software that conformed to its local standard operating practice manual in using GO TO DEPENDING ON to send the flow of control to appropriate routines, such as taxing income from irrigated farming, based upon transaction codes. However, the software had no domain checking of the transaction code, and did not have any programmatic trap on the transaction code. The result sometimes was that ‘impossible errors’ would appear in the output for no obvious reason, or that an operation ‘runaway’ would happen, sometimes harmlessly but more often resulting in corrupted data. A temporary fix inserted a PERFORM bypass routine to write all questionable data to an error report. Manual review of that report generated corrected data for reprocessing on later runs. The result was that user complaints about ‘impossible errors’ dropped sharply.

3.9. Input and output verbs

The COBOL input and output verbs CLOSE, OPEN, READ, WRITE, REWRITE, START, and DELETE have some error checking associated with them for determining whether or not the action was successfully completed. For example, executing a READ causes the runtime environment to access hardware traps to update the value of FILE STATUS provided that it was included in the file control entry (Pollack and Sterling, 1993).

A general fix requires not only activating FILE STATUS but also establishing a named section in the COBOL declaratives section. Then the programmer can insert there a USE AFTER STANDARD EXCEPTION PROCEDURE or USE AFTER STANDARD ERROR PROCEDURE to get action from the runtime environment. In addition, the programmer can insert IF statements immediately after the input–output verb in the regular part of the program to examine the value of FILE STATUS in detail and select additional processing to be done or skipped over, as appropriate. Some job sites make such FILE STATUS routines be their local standard, and copy the relevant source code into the programs.

For READ, the only quick fix is limited to the use of the optional AT END phrase available with some input and output verbs. For WRITE, the only quick fixes are the use of the optional INVALID KEY or AT END-OF-PAGE phrases. For DELETE, REWRITE, and START, the only quick fix is the use of the optional INVALID KEY phrase.

An international bank had, and enforced, good standard practices on identifying FILE STATUS errors, but did not include upgrading them when it converted from COBOL-74 to COBOL-85. The result was that the new COBOL-85 status codes, such as for logic errors (4x), were not tested. As the first part of a fix, all of the bank’s standard input/output error routines were updated to conform to the COBOL-85 standard. Many programs then abended with code 43 (required READ not successfully executed) and code 46 (no valid NEXT RECORD established). Correcting these improved the accuracy of the output data and reduced computer run irregularities.

4. CORRECTING DEFENSIVE TRAP USAGE

4.1. Acts of commission

Fixing omissions of defensive traps improves software quality, but leaves untouched correcting acts of commission in the use of defensive traps. In effect, the acts of commission are the result of changed requirements, oversights or errors in either the design or implementation of the software. The net result is usually just that the existing software is wrong in its use of defensive traps—that is, the traps present are misused or fail to provide the full measure of defensive trap action intended.

Some of these misuses or weak uses of defensive traps centre on situations unique to COBOL, such as some uses of the COBOL verb PERFORM. Others centre on common areas of concern to analysts and programmers in many languages, such as subscripts. Usually the fixes that can be made during software maintenance require more extensive work than do the fixes for omissions of defensive traps. They are also more likely to require attention to preserving the integrity of the implementation of the 'business rules' embodied in the program, and hence more likely to require the maintenance personnel to interact with the users of the software.

4.2. Index and subscript domain checks

While a few vendors of COBOL compilers include 'range checking' capabilities in their COBOL extensions, standard COBOL does not include such a capability directly. Hence, to check the values of indexes and subscripts, the programmer or analyst usually must write source code to get the checking done, although COBOL's DEBUG can help (Vesely, 1991, pp. 105–109). The usual choice involves the use of IF, as in:

```
IF index-subscript 〈 1 OR 〉 maximum-index-subscript ...
```

Processing with out-of-domain index or subscript values can produce subtle 'impossible errors' that are often hard to track down. For example, when the computer executes

```
MOVE who-knows-what (out-of-domain-index) TO numeric-field
```

the index value causes the sending data *not* to come from the intended source. Since the intended destination is a numeric field, the sending field should also be numeric. With an out-of-domain index, the sending field might be alphabetic, thus filling a numeric field with non-numeric data. This is a time bomb waiting for Murphy's Law to tell it when to explode. It can even be a transported time bomb if this program writes the 'numeric' field to a file that is processed by some other program later. This later program may fail, leaving analysts and programmers to wonder how the 'numeric' field got corrupted! In such later programs, it may be necessary to insert a temporary work around, either by adding domain checking to the program, or by writing the symptom data to an error file, as shown below, and bypassing further use of the corrupted data, as by invoking a series

of PERFORM statements or executing a GO TO in a non-structured manner. Assuming a success–failure code has been established, such as in a field current-trancode, then one way of saving the symptom data is as follows:

```
IF current-trancode > good-trancode
    MOVE relevant-data TO symptom-report-area
    WRITE symptom-report
END-IF
```

The postal service for a country was suffering from 'strange errors' in its registered mail tracking system. The programs were written in IBM VS COBOL II (an extension of standard COBOL-85) which has a compiler option for checking subscript ranges. Turning on this option (SSRANGE) caused all but a couple of the system's programs to abend. Recoding was then done to resolve the causes of the 'strange errors'.

4.3. IF NEGATIVE

An easy way to create an instance of data corruption in COBOL is to use MOVE to copy a signed numeric value into a field defined for unsigned numeric values. Presto! If the number was negative, in received form it becomes positive! A more subtle method is to use SUBTRACT operand-A FROM operand-B when operand-A is greater than operand-B and signed, but operand-B is not defined as having a sign. Of course, it is always possible that some 'tricky' programmer has used either of these techniques deliberately to produce the absolute value of a number. More likely, however, these are bugs and just instances of weakness in programming. Accounting reports containing strange results are sometimes symptoms of such bugs. A good defensive trap is to insert an IF NEGATIVE test before a MOVE or SUBTRACT involving operands that are defined with different signings. A better treatment is to correct the signing, if the logic of the 'business rule' being implemented permits it. The COBOL DEBUG capability can be used to help locate such signing problem areas (Vesely, 1991, pp. 137, 182–199).

An international manufacturer of lighting fixtures for business use, used a COBOL-implemented system for manufacturing control, including inventory control. But the inventory control part appeared, on average, to be showing more stock on hand of parts and subassemblies than it should. The manufacturer's CEO *knew* something was wrong with the system, for it never showed any negative inventory balances. Negative balances were possible and expected because the manufacturer accepted orders without inventory on hand, on the theory that the manufacturer could buy or build the needed items in time. The programs in the system had no IF NEGATIVE traps. These were inserted. Subsequent runs showed that many programs moved negative values to unsigned fields, thus causing the minus signs to be lost, and a few programs subtracted signed fields from unsigned ones. To meet the need for a good fix, the data divisions were changed to sign all fields used in arithmetic operations. When the CEO saw the resulting inventory data, he stopped shouting at the Information Systems organization.

4.4. Overlapping operands

The COBOL Standard in the subsection ‘Overlapping operands’ in the section on ‘Execution’ in the chapter on ‘Procedure division’ states as follows (ANSI, 1985, p. VI–69): ‘When a sending and a receiving item in any statement share a part or all of their storage areas, yet are not defined by the same data description entry, the result of the execution of such a statement is undefined. In addition, the results are undefined for some statements in which sending and receiving items are defined by the same data description entry’. The meaning of ‘undefined’ was noted earlier in this paper in the section on arithmetic verbs. Some overlapping operands are intentional and work as expected, as for example, adding a number to itself by ADD sum-field TO sum-field. Some overlapping operands result in data corruption.

Overlapping operands are most often found in arithmetic, INSPECT, MOVE, SET, STRING, UNSTRING statements, and may result from the use of REDEFINES or RENAMES in the data division. Compilers and runtime environments for COBOL have varied responses to overlapping operands, such as:

- warning messages of various severity levels during compilation,
- object code that sometimes abends in execution,
- object code that does not abend but throws a runtime warning message, or
- compilation and execution with no messages (the most common response).

There are no quick fixes for unintentional overlapping operand usage. The usual temporary work around is to write an error message and then bypass, as discussed for arithmetic verbs. The best corrective fix is usually to examine, by desk debugging, the usage of all operands involved in each RENAMES and each REDEFINES with the aid of a compile listing including an xref. Some software tools have analysis features that can be useful in locating overlapping operands, such as Jagadeesh (1993).

4.5. PERFORM

The way that the COBOL PERFORM actually works is one of COBOL’s best kept secrets (Crawford, 1990). Nearly all COBOL analysts and programmers act as though they believe that using an EXIT paragraph after a performed paragraph (i.e., a COBOL paragraph executed via a PERFORM) results in a return of control. Some liken it to the automatic return of control associated with the use of CALL. The coming new standard for COBOL clarifies the connection between EXIT and PERFORM (Chapin, 1997, pp. 105, 117–119, 353), whereas the older standard, in its section on EXIT (ANSI, 1985, p. VI–88) simply says: ‘An EXIT statement serves only to enable the user to assign a procedure-name to a given point in a program. Such an EXIT statement has no other effect on the compilation or execution of the program’. The execution of a PERFORM includes acting as though it places a temporary (single-use) GO TO next-executable-statement-following-the-perform-statement at the end of the set of performed statements. Whether or not the analyst or programmer used an EXIT is irrelevant to the PERFORM, even though the EXIT must be in a paragraph by itself and is typically used to flag, for human attention, the logical end of a set of statements to be executed via a PERFORM.

Two kinds of trapable difficulty can arise with PERFORM, both related to overlapping of control flows. Figure 2 illustrates, in skeleton form extracted from COBOL source code, one situation involving the interaction in execution of three statements, a PERFORM, an EXIT and a GO TO. The single lower-case letters represent COBOL line numbers and the periods as line numbers represent non-relevant but present lines of executable statements.

In Figure 2, line r is the spot where, in the compiled object code during execution, the equivalent of a temporary GO TO line-n gets generated when the object code for line m is executed. But when A is ‘true’, that transfer of control never gets used, and lies in wait. If Paragraph-example should later happen to be executed by means of a GO TO or fall-through of control into line o with A false, then upon completion, control will go to line n. That line might or might not be the correct place for the control flow to go, and hence this code contains a latent bug.

Figure 3 offers a second example, one involving only PERFORM, this time nested in an overlapping manner. Again, lower-case letters and periods in the line number column represent line numbers.

In Figure 3, Para-h gets executed by and is included within the scope of the two PERFORM statements, the one at line x and the one at line c. Since line x executes first, line l is the place where the implicit GO TO line-y gets set. Then at line c, line p is the place where the implicit GO TO line-d gets set. Hence, Para-m never gets executed, because the PERFORM at line c causes line y to be executed after line k and because the PERFORM from line x is still active (not completed). Para-h only gets executed once, not twice, and the flow of control gets back to line y unexpectedly early.

Quick fixes do not take care of such difficulties adequately. For example, inserting an EXIT statement in its own paragraph can serve as a flag to analysts and programmers

```
.
.
m        PERFORM Paragraph-example
n        next-executable-statement
.
.
o    Paragraph-example.
p        some-executable-statement
q        IF A true GO TO Beyond-exit-paragraph
r
s    Paragraph-example-exit.
t        EXIT.
.
.
u    Paragraph-more.
.
.
v    Beyond-exit-paragraph.
.
.
```

Figure 2. First example of overlapping control flows

```
.
.
x       PERFORM Para-a THRU Para-h
y       next-executable-statement
.
.
a   Para-a.
b       an-executable-statement
c       PERFORM Para-h THRU Para-m
d       an-executable-statement
.
e   Para-e.
f       an-executable-statement
g       an-executable-statement
.
h   Para-h.
i       an-executable-statement
j       an-executable-statement
k       an-executable-statement
l
m   Para-m.
n       an-executable-statement
o       an-executable-statement
p
q   Para-q.
.
.
```

Figure 3. Second example of overlapping control flow

reading the source code, but has no effect upon execution in trapping either an inadvertent fall-through or an inadvertent GO TO or PERFORM transfer of control. For insight and aid in identifying the misuse of PERFORM and EXIT, a defensive aid is to insert counters at the entrance to and exit from each PERFORM—mismatches indicate trouble to be likely. A good defensive trap can be written but is too lengthy to cover briefly here (Vesely, 1991, pp. 232–255).

A medium-size insurance company was facing a limitation on its in-force growth—its home-grown three-year-old premium accounting system running on its mainframe computer was taking too much time out of the company's normal three-shift computer operation. A consultant specializing in computer performance improvement was engaged to speed up the computer's performance on the premium accounting system. The consultant's investigation revealed nothing specific causing slow performance except that most of each of the programs of the system were being executed in handling most transactions. In digging deeper, the consultant inserted execution counters as defensive aids into the programs to tally the entrance into and exit from each PERFORM. An analysis of the resulting counts showed that overlapping PERFORMs were numerous and were resulting both in redundant and erroneous processing. This additional processing not only made the system run slowly but also was gradually corrupting the files and generating subtle errors in the output reports. The company management was shocked at the evidence of bad data, and dismayed at the amount of maintenance that would be needed to correct the

programs, the files and the associated accounting procedures. In reaction, management elected to junk the system in favour of licensing a premium accounting package touted as offering lots of capacity for handling growth.

4.6. Qualification

COBOL permits the use of generic data names (provided they are not reserved words) for identifying data. Figure 4 gives a short example, presented in two columns to save space.

For example, when the analyst or programmer uses a name defined more than once in the data division, the definitions do not have to be the same, and may be very different. In writing statements in the procedure division, however, the analyst or programmer in such cases has to do more than just specify the data name in order to avoid ambiguity for the compiler. For example in the context of Figure 4, the following is ambiguous: `MOVE street-address TO customer-street`. Is it the student's street address, or the instructor's street address? To clarify, the analyst or programmer must add qualification, such as: `MOVE street-address OF student TO customer-street`. Qualification can apply to paragraphs in sections and to text names in libraries, as well as to data names.

Most COBOL compilers warn about needed qualification, but produce object code anyway by assuming a qualification. This may not be what the analyst or programmer intended, and during maintenance, the compiler-generated warnings may not be available. The quick fix is to add qualification on any instance that generates unwanted data during execution, and recompile checking for warnings. The better fix is to use an xref sorted in order by name, and check all entries with multiple definitions.

5. CONCLUSION

Re-engineering, CASE tools, debugging tools and 4GLs have been used for over a decade. Their protagonists have claimed their use would lead to the replacement or restructuring of COBOL legacy software. But the sad truth has been that most old COBOL programs just get older as they stay in production use. Both those programs and many COBOL programs modified with the aid of CASE and re-engineering, do occasionally produce incorrect results and 'impossible' errors that baffle software maintenance personnel. This paper has presented field-proven practical techniques for installing defensive traps into existing production COBOL programs to isolate these errors. Inserting and strengthening defensive traps during software maintenance can improve the quality of the software.

```
01 Student.                    01 Instructor.
    03 Name.                       03 Name.
    03 Street-address.             03 Street-address.
    03 Telephone.                  03 Telephone.
```

Figure 4. Example of generic names needing qualification

References

ANSI (1985) *Programming Language COBOL*, American National Standards Institute, New York, NY, 533 pp.

Chapin, N. (1997) *Standard Object-Oriented COBOL*, John Wiley & Sons, Inc., New York, NY, 394 pp.

Crawford, M. A. (1990) 'Lurking within COBOL PERFORMs', *Journal of Software Maintenance,* **2**(1), 33–46.

Jagadeesh, J. M. (1993) 'A trap for hard-to-find bugs', *Computer*, **26**(11), 66–68.

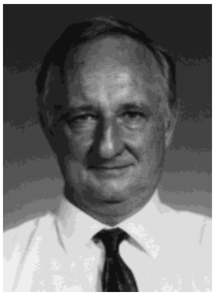
Pollack, S. V. and Sterling, T. D. (1993) 'Trap', in Ralston, A. and Reilly, E. D. (Eds), *Encyclopedia of Computer Science*, Van Nostrand Reinhold, New York, NY, p. 1390.

Vesely, E. G. (1989a) *COBOL: A Guide to Structured, Portable, Maintainable, and Efficient Program Design*, Prentice-Hall International, Englewood Cliffs, NJ, 410 pp.

Vesely, E. G. (1989b) 'Maintenance and syphilis', *American Programmer,* **2**(1), 17–21.

Vesely, E. G. (1991) *COBOL Debugging Diagnostic Manual*, Prentice-Hall International, Sydney, Australia, 324 pp.

Author's biography:



Eric Garrigue Vesely works as a Consultant and Principal for a computer services company headquartered in Malaysia. Eric's maintenance speciality is legacy systems implemented in COBOL. From his experience, he has prepared a software tool, CARE (for Computer Aided Re-Engineering), that is available on floppy disk and published as part of his 1991 book. Dr Vesely is the author of four books on COBOL programming, and has served as a member of the Advisory Editorial Board for the *Journal of Software Maintenance*.

